MSc in Computer Science and Engineering

Software Engineering 2 Project

TrackMe Software Design Document

Authors :
Andrea Biscontini - 901310
Marco Gelli - 901470
Alvise de'Faveri Tron - 920882

Professor:
Elisabetta di Nitto

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to give a detailed description of the TrackMe system's design which has been developed starting from the requirements, defined in the RASD. The next sections will focus on the implementation of the system, describing more precisely the architecture, the components, the deployment and the run-time processes that identify it. The main goal is to provide an overall analysis of the product architecture.

With respect to the RASD, this document is addressed more specifically to the developers, managers, testers and system administrators that will have to implement, manage and maintain the system.

Moreover, this document will provide the reader with in-depth details of all the internal and external services that the system offers and needs in order to reach the given goals, with a particular attention for the responsibilities of each component and interface.

1.2 Scope

The system is made of three sub-systems: Data4Help, AutomatedSOS and Track4Run. Since each of them has a different purpose, their internal characteristics also reflect this difference. Nevertheless, the external interfaces of each subsystem have been accurately designed to minimize the potential inefficiencies when the whole system is pulled together.

In particular, Data4Help provides a platform for sharing personal data be-

tween users and third parties, and this must be done in a secure and reliable way: these issues lead the design of the whole architecture of this subsystem.

On the other hand, AutomatedSOS monitors its users state of health and, in case of emergency, calls an ambulance. Bearing this in mind, the architecture of this subsystem will be developed to be as responsive as possible and minimize all the possible delays.

Finally, Track4Run gives the possibility to organize, join and watch run events. For this reason, user-friendliness and user experience are the main concerns of this subsystem, which will be developed to be easy to access and navigate in all its many features.

1.3 Definitions, Abbreviations and Acronyms

1.3.1 Acronyms

- **RASD:** Requirement Analysis and Specification Document
- **TP:** third party
- **DS:** data source
- **API:** Application Programming Interface
- **SDK:** Software Development Kit
- **DB:** Data Base
- **MVC:** Model-View-Controller
- **JSON:** JavaScript Object Notation
- **JWT:** JSON Web Tokens
- **REST:** Representational State Transfer

1.3.2 Abbreviations

- **Gn**: n-th goal
- **Dn**: n-th domain assumption
- **Rn**: n-th functional requirement

1.4 Revision history

Revision	Date	Changelog
1.0	10/12/2018	First document issue

1.5 Document Structure

- **Chapter 1** Gives a general description of the document, its purpose and an overview of the product's characteristics. It also specifies some text conventions used throughout the document.
- **Chapter 2** This section provides an inside view of the system, its components and the deployment of its parts. Also a description of the architectural patterns and design decisions is given, in order to provide a detailed view of the whole system.
- **Chapter 3** This chapter refers to section *3.1.1 - User Interface* of the RASD.
- **Chapter 4** Here we will explain how the goals stated in the RASD are met by our architecture by providing a traceability matrix.
- **Chapter 5** Here we identify a priority for the development and integration of each component in order to understand which activities should be carried out in parallel and what are the dependencies existing between the components.

2 Architectural Design

2.1 Overview

As stated above, each subsystem has to accomplish its own goals. In particular, we can consider AutomatedSOS and Track4Run as simple applications for smart-devices, while Data4Help as a service provider that should implement a more complex, scalability-oriented architecture.

The figure below represents the physical architecture of the system to-be.

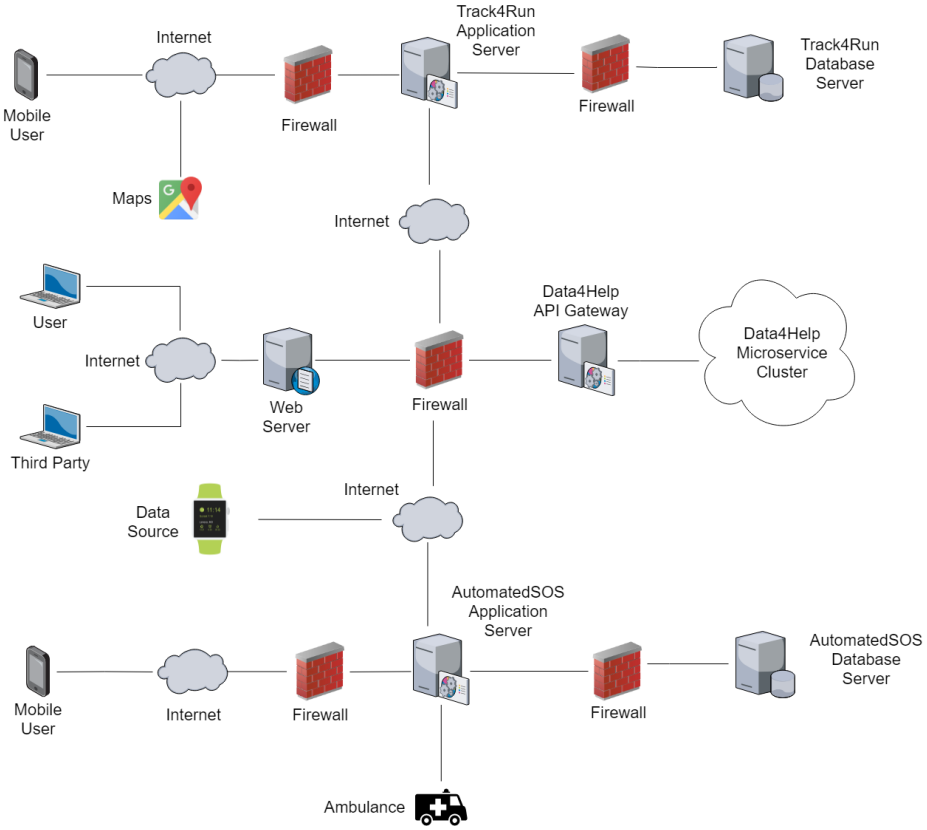


Figure 1: General Architecture

As shown in the figure, Data4Help services can be accessed through different channels. In particular, we provide a web interface for the end users and an

API for data sources and third party applications.

Since the data sources will continuously provide the system with real-time user data, we expect that this flow of information will soon grow into a huge number of requests. For this reason, this subsystem has to be designed to guarantee high scalability and to ensure a high availability level even in extreme conditions. In particular, a microservice-based architecture coupled with message queues and scalable databases should be adopted, as we will describe in the following sections. Finally, an API Gateway is used to access all Data4Help interfaces, in order to easily manage the authentication phase and load balancing from the very first moment of any request.

On the other hand, AutomatedSOS and Track4Run are designed as three-tier applications with a thick client (i.e. smart-phones and smart-watch applications), a lightweight back-end server and a storage module. In the design of these two subsystems we applied the separation of concerns principle for different reasons:

1. Modularity: AutomatedSOS and Track4Run business logic are separated from Data4Help business logic.
2. Back-end performances: we ensure a fast and reliable service leaving the presentation part to the mobile application.
3. System maintainability: multi-layered architecture and the achieved modularity helps long term maintainability.
4. Mobile devices performances: considering that CPU usage and power consumption are significant issues, we lighten their workload.

2.2 High Level Architecture

From the component point of view, each subsystem can be divided into *back-end* components, which are responsible of carrying out the business logic of the subsystem, and *front-end* components, which provide a presentation layer to the end users and SDKs to the external developers who want to

access the system's services. Finally, we rely on some external components to provide some of the services offered by the system.

In the following diagram, the main components and interfaces are highlighted, to give a general idea of the system design. Components and modules represent a set of services grouped together, and the internal interactions between modules are not shown for sake of simplicity. This description will be done in the following sections.

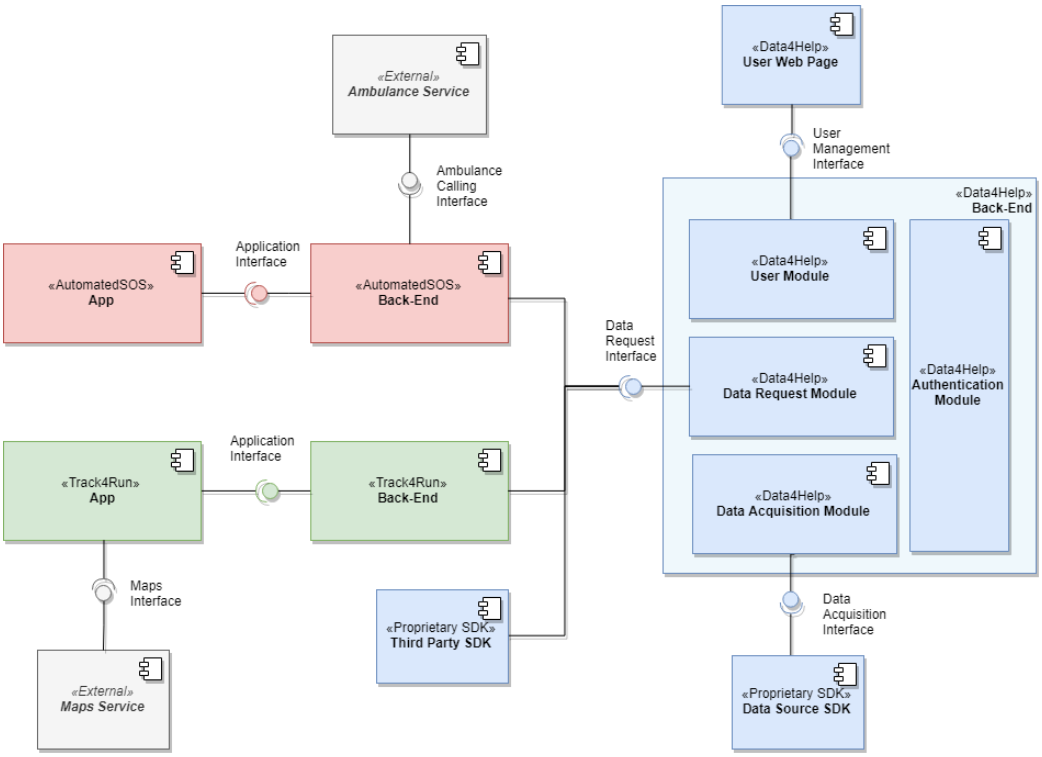


Figure 2: High Level Components

As can be seen, the Data4Help back-end consists of four major modules:

- **User Management**: provides an interface for managing the user account and configure his data sources. This interface is used by a *Web Application* that can be accessed via web browser.

- **Data Acquisition**: provides the interface for collecting data from data sources. A proprietary *Data Source SDK* is given to the external developers to integrate the use of this interface into their software and send data from external devices to Data4Help.
- **Data Request**: enables third parties to make requests on the data received by the subsystem. This interface is used by *Track4Run* and *AutomatedSOS* back-ends, but it can also be accessed by any other third party via the dedicated *Third Party SDK* given to their developers.
- **Authentication**: enables the system to recognize and authenticate incoming requests and associate them to a specific user, third party or data source.

The other two back-ends are mainly responsible of exploiting Data4Help's services and offer an interface to the mobile application. In particular:

- **AutomatedSOS App Interface**: provides the functions to log in a user, modify its threshold and monitor its health status.
- **Track4Run App Interface**: provides the functions to log in a user, create and edit a run, search for scheduled events, join them and watch runners during a competition.

Finally there are the external services: for AutomatedSOS, an external Ambulance Calling Interface is needed by the back-end to fulfill its goals; Track4Run instead needs a Maps Service such as *Google Maps* that should be accessed directly from the application to minimize latency and bandwidth occupation in the communication between client and server.

Another definition of the services offered by the system can be found in the figure below, which highlights the dependencies between the Data4Help system and its actors providing a list of the services used internally and externally by the system.

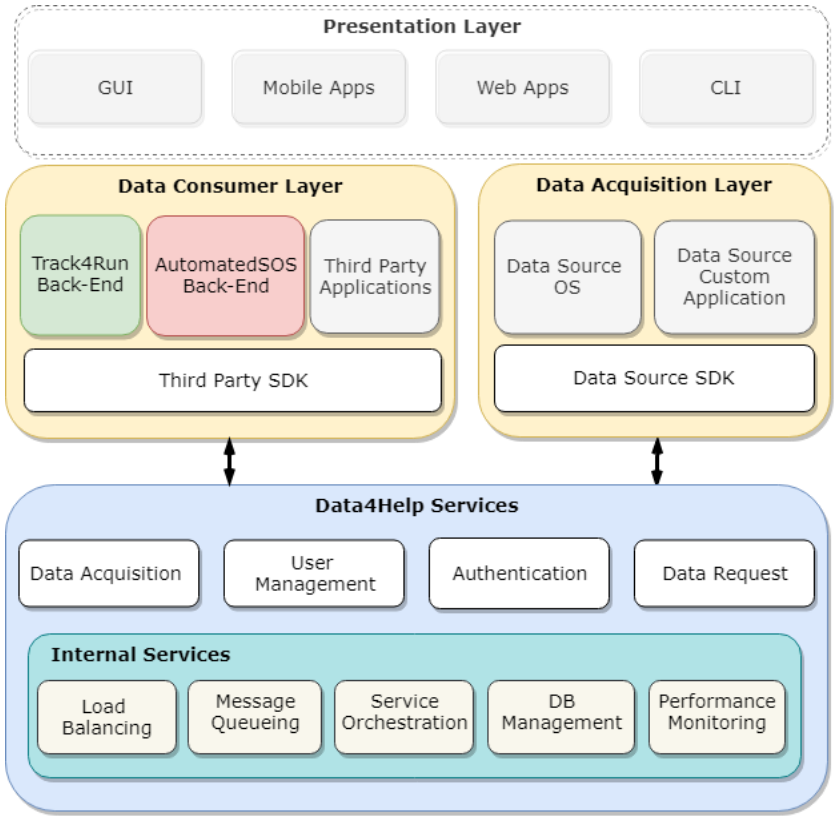


Figure 3: System Layers Architecture

The whole system's internal and external services can be divided into five layers:

- **Data4Help Internal Services Layer**: these components are used internally by Data4Help to provide highly available services. They mainly deal with microservices management and will not be discussed in detail, since there are many available options on the market such as Docker and Kubernetes for container management, MongoDB for data management, RabbitMQ for message queueing, Prometheus for monitoring etc...
- **Data4Help External Services Layer**: this layer provides the four main functionalities of Data4Help, which have been discussed previ-

ously.

- **Data Acquisition Layer**: this layer is built upon Data4Help Services. It contains the SDK on which external data sources can rely on to send collected data to Data4Help.
- **Data Consumer Layer**: also this layer is built upon Data4Help Services and contains the SDK to send API requests used by third parties as well as AutomatedSOS and Track4Run back-ends, which can be seen as *consumers* of the Data4Help services.
- **Presentation Layer**: built upon the previous layers. It is the mobile application of AutomatedSOS and Track4Run which provides a GUI to the end user.

2.3 Component View

2.3.1 Data4Help

Here is a detailed view of the components of the Data4Help subsystem: each service must be considered as independently deployed and can be replicated in many instances, following the typical microservice cluster architecture.

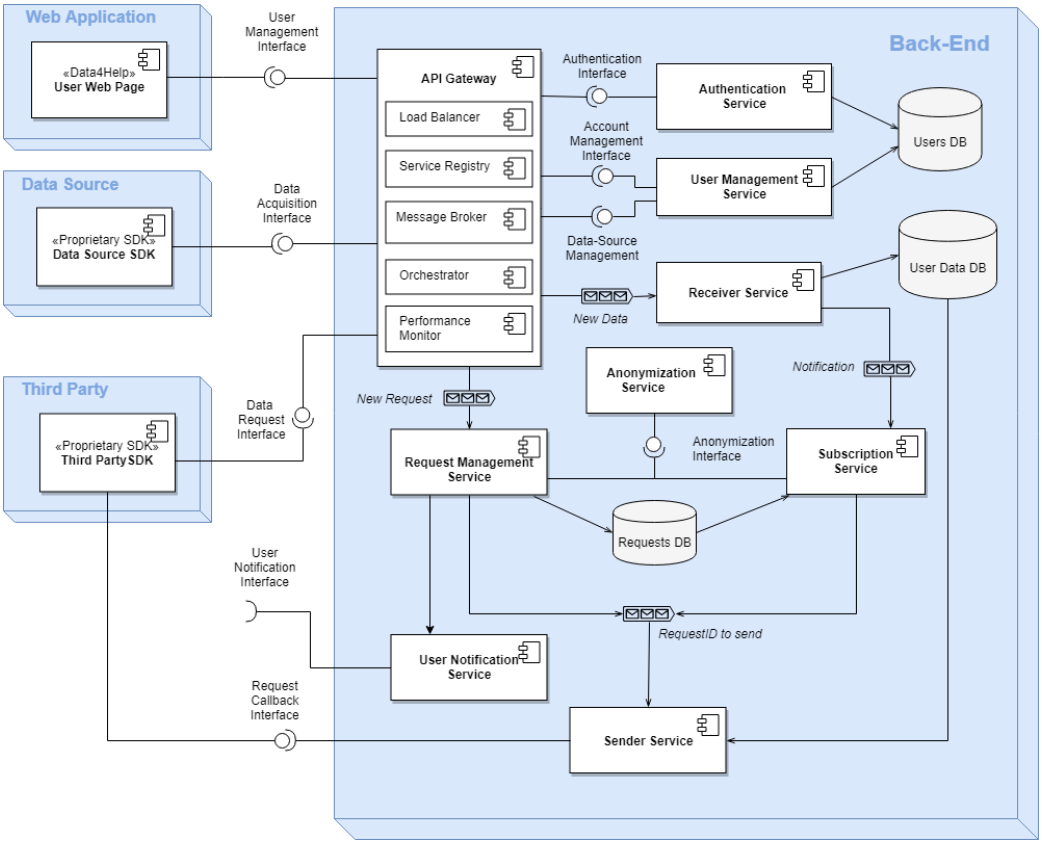


Figure 4: Data4Help Components

The components of this subsystem are:

- **API Gateway**: This is the single entry point through which all other services can be accessed. For more information see *API Gateway Pattern* in *Section 2.7*. In this diagram, other components are also represented in the API Gateway:
  - *Load Balancer:* realizes the balancing between different instances of the same services, deciding where to redirect each request.
  - *Service Registry:* keeps track of all the deployed services instances.
  - *Service Orchestrator:* responsible of managing the deployment of new services and the failure of existing ones.

  - *Message Broker:* enables messaging between services.
  - *Performance Monitor:* continuously tracks end-to-end delay of our services and notifies the need for more resources. This component is needed to achieve high responsiveness of the system, in particular this is required to meet AutomatedSOS 5 seconds time constraint.

  Please notice that, for the sake of simplicity, the interaction between these components and the rest of the system is not represented here. As already mentioned, all these services are part of the internal services layer and are considered to be something already existing during the development of this subsystem.

- **Authentication Service:** in charge of producing tokens when new users register and verifying the incoming ones when requests are made to the API Gateway.

- **User Management Service:** provides an interface for the web application. The main functions are account management (password changing, name setting etc.) and data source management (i.e. selection of a data source for each tracked parameter).

- **Receiver:** responsible of managing a new incoming data source packet, verifying the user preferences for that data source and eventually saving the packet in the User Data DB. It is also responsible of notifying the subscription service with the ID of the received parameter, so that the interested third parties can be notified of the new data.

- **Request Manager:** responsible of receiving and parsing the data requests coming from third parties. In case of one-shot requests, a user notification service will be called to ask user's permission, and then the sender service will be invoked to send the actual response. If the request is instead a subscription request, this will be saved in the Requests DB after user's approval. Group requests don't need user approval, but they need to be filtered through the anonymization service.

- **User Notification:** provides a way to notify the user through different communication channels (e.g. e-mail, SMS, in-app notifications etc.) and receive his response.

- **Sender:** in charge of consuming the sender queue, build the request for each response and send it to the corresponding callback interface.
- **Subscription Service:** in charge of consuming the notifications received by the receiver service(s) and control if the new data fits into a single user or group data subscription. In this case, the corresponding response will be queued in the sender queue.
- **Anonymization Service:** simply tells if a given group request can be properly anonymized. It should be accessed using a synchronous protocol.

2.3.2 AutomatedSOS

AutomatedSOS follows the typical client-server architecture: in the back-end we have all the business logic of the application, while the presentation is carried out by the Client Application.

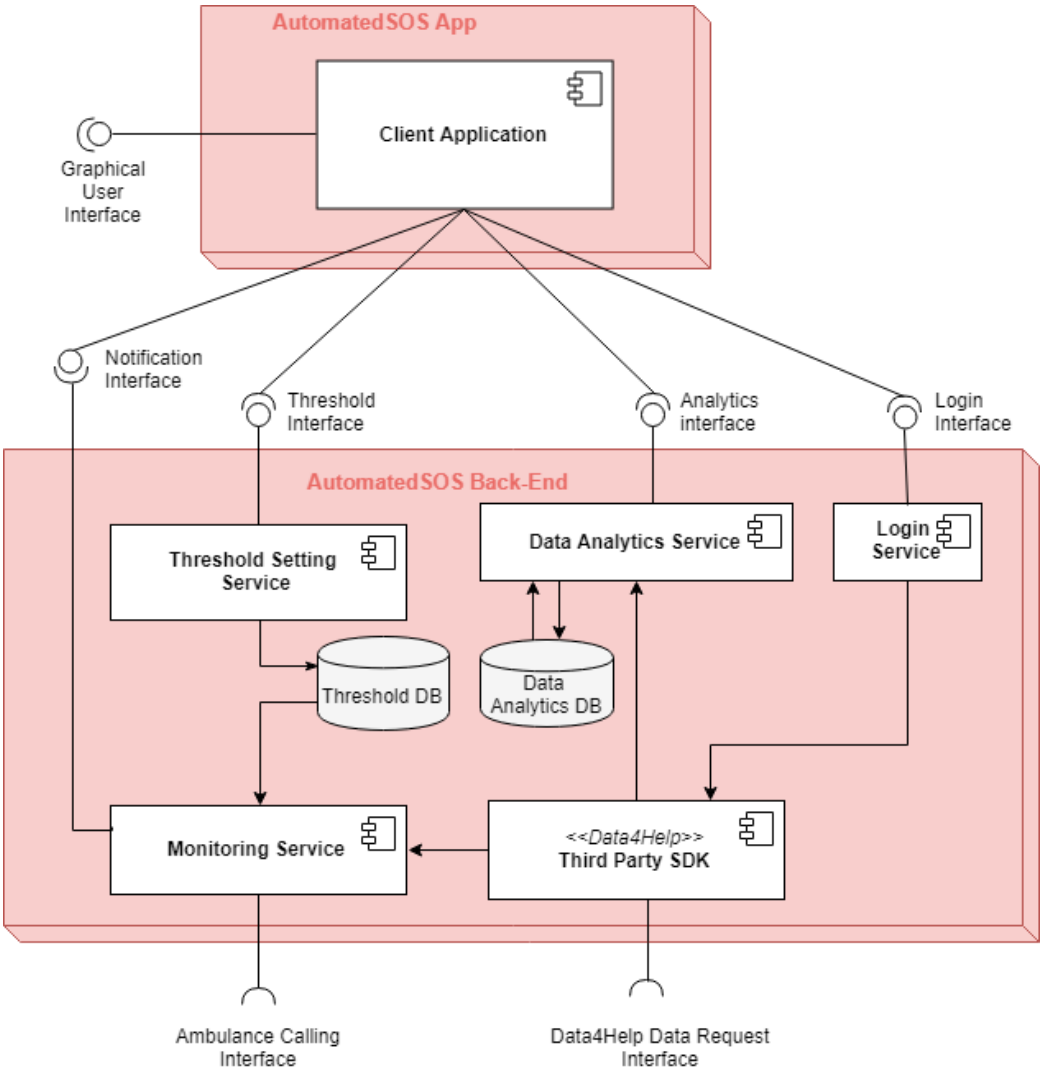


Figure 5: AutomatedSOS Components

- **Client Application:** application on user's device responsible of providing a graphical interface to the user and interacting with the back-end services of AutomatedSOS.
- **Threshold Setting Service:** gives to the user the possibility to change the default threshold values of his tracked parameters and save them in the Threshold DB.

- **Data Analytics Service:** allows the user to check the aggregated statistics about the data collected in a given time-lapse with a given granularity.
- **Login Service:** responsible of authenticating the user with his Data4Help credentials using the Third Party SDK component. It also has to subscribe to the user's new data.
- **Monitoring Service:** receives the data coming from Data4Help through the SDK and, whenever a threshold is exceeded, interacts with the external Ambulance Calling Interface, communicating the location of the user in danger. The service gives priority to the data coming from users that are near to their thresholds using a sorted queue.
- **Third Party SDK:** communicates with Data4Help using his Authentication and Data Request interfaces.

2.3.3 Track4Run

Track4Run also follows the typical client-server architecture: in this case, the Client Application interacts not only with the back-end but also with an external Maps Interface.

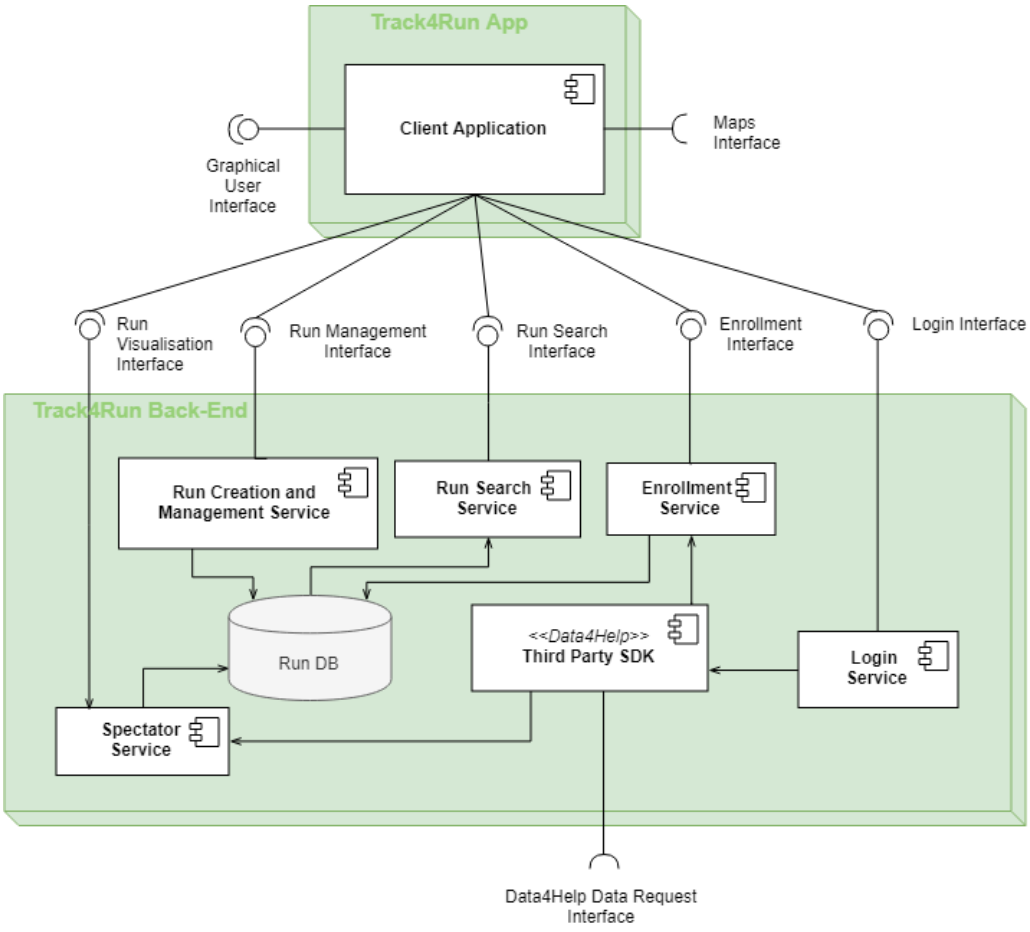


Figure 6: Track4Run Components

- **Client Application:** application that runs on user's device responsible of providing a graphical interface to the user and interacting with the back-end services of Track4Run. It also interacts with an external Maps Interface in order to retrieve the maps that will be used by the application.
- **Run Creation and Management Service:** gives the user the possibility to create, edit, delete and extend ownership of a run event by interacting with the Run DB.
- **Run Search Service:** gives the user the possibility to browse all the

created run events stored in the Run DB.

- **Spectator Service:** responsible of retrieving runners position during the run and then forwarding it to the Client Application which in turn will display the positions of the runners on a map.
- **Enrollment Service:** allows the user to enroll in a created scheduled run event. In this way he will be added to the list of participants.
- **Third Party SDK:** communicates with Data4Help using his Authentication and Data Request interfaces.
- **Login Service:** responsible of authenticating the user with his Data4Help credentials using the Third Party SDK component. It also has to subscribe to the user's new data.

2.3.4 Entity-Relationship Diagram

The following diagram provides a graphical representation of the data entities of the system. Different colours represent the three different subsystems.

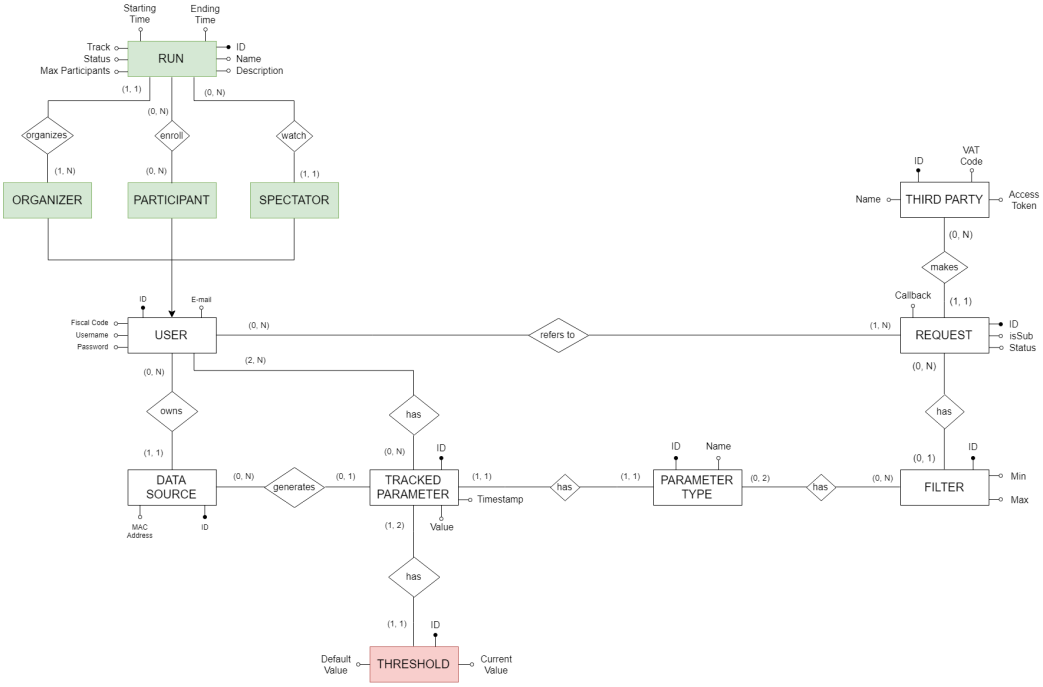


Figure 7: Entity-Relationship Diagram

2.4 Deployment View

The deployment of the three subsystems reflects the architecture described in the previous sections. AutomatedSOS and Track4Run follow the client-server pattern, with a lightweight back-end (e.g. Servlet) and a DB server (SQL or NoSQL).

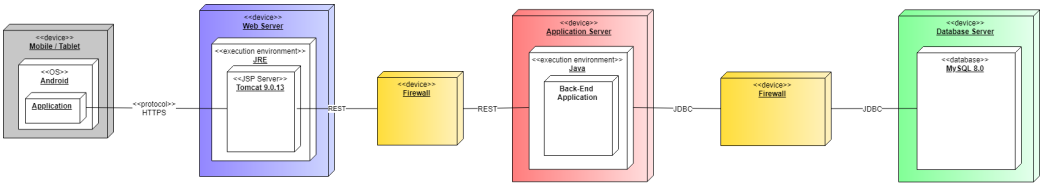


Figure 8: AutomatedSOS and Track4Run Deployment Diagram

On the other hand Data4Help follows the microservices architecture pattern. In particular, the API Gateway and Server-Side Discovery patterns are ap-

plied: an *API Gateway*, deployed on a dedicated machine, will filter all the incoming requests and forward them to the single services. To know where these services are located, a *Service Registry* has to be deployed at a known address. Each back-end service is instead deployed in a separated *container*, located in a physical or virtual machine, so that a new instance of a service can be deployed in a new container when needed. On the side of this microservice cluster, entities such as a Container Orchestrator, Load Balancer, Performance Monitor and a Message Broker are needed to properly manage the cluster. These have to be deployed in containers inside dedicated and reliable machines that are part of the internal network, to achieve scalability.

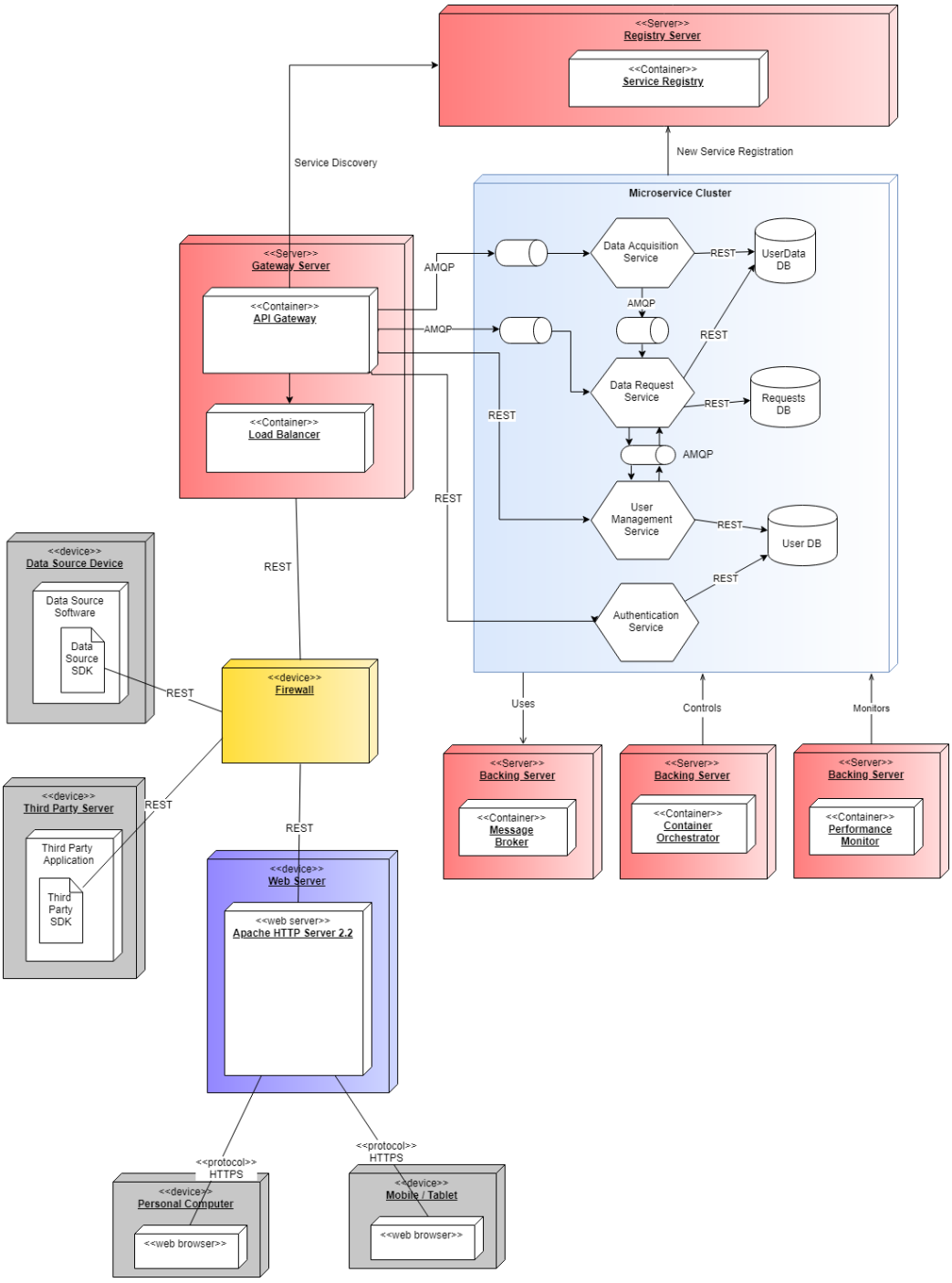


Figure 9: Data4Help Deployment Diagram

2.5 Runtime View

Many sequence diagrams have already been added in the RASD to show the interaction between the system and the users. Here some other internal details are provided for the four main functions of the Data4Help subsystem.

2.5.1 Authentication

Data4Help's authentication flow is taken from the classic token authentication pattern. Any *Service Consumer* (i.e. end-users, third parties and data sources) can access the internal services if and only if they provide an access token previously generated by the system, during a login or registration phase.

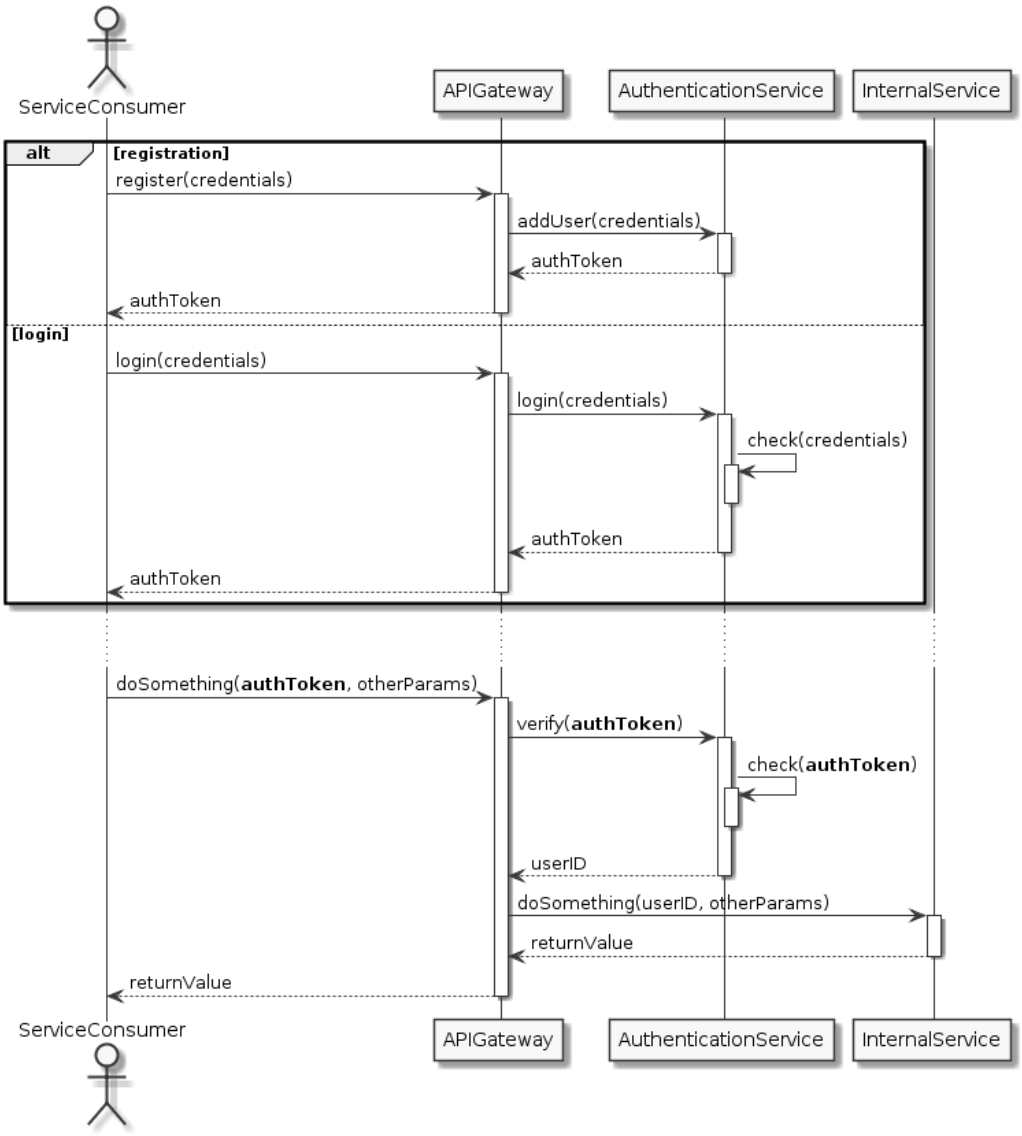


Figure 10: Authentication Sequence Diagram

2.5.2 Data Acquisition

Each time a new data is sent to the system, a Receiver must process that request and save the corresponding data on the User Data DB. Thereafter,

an asynchronous notification will be sent to the Subscription Service to signal that a new data is ready to be sent. When the Subscription Service is ready, it starts handling the notification: first it will check if some third party is subscribed to that specific user's data. Then, it will check if the new data is part of a data group and send the whole group's data to the subscribed third party if they can be properly anonymized.

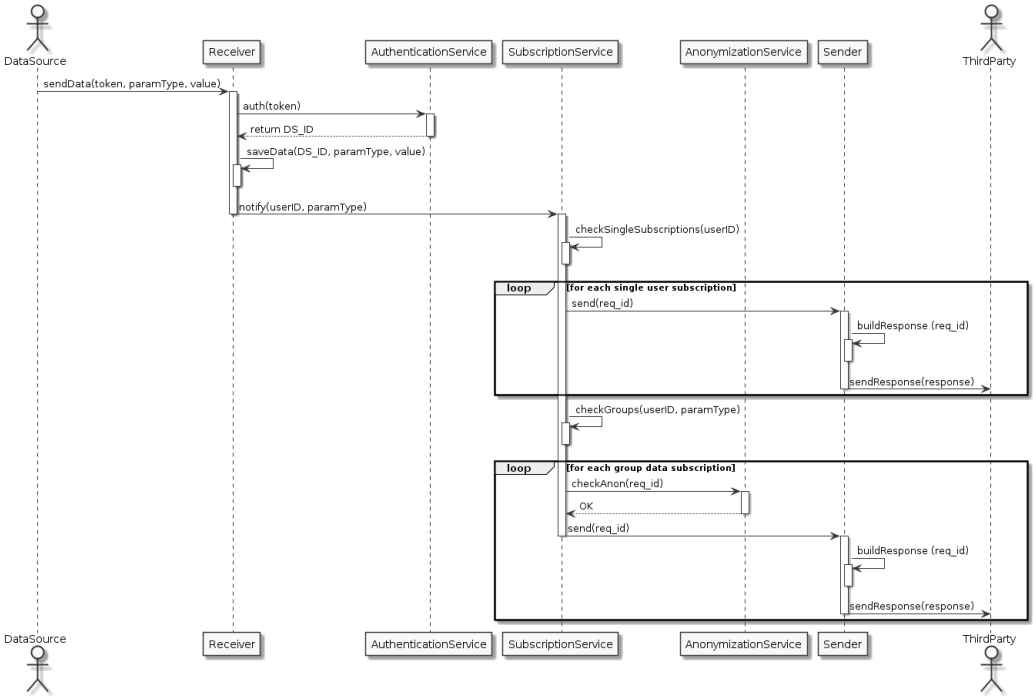


Figure 11: Data Acquisition Sequence Diagram

2.5.3 Single User Data Request

In case a third party performs a single user data request, this request is marked as *pending* until the target user approves it. This is true both for one-shot requests and subscription requests.

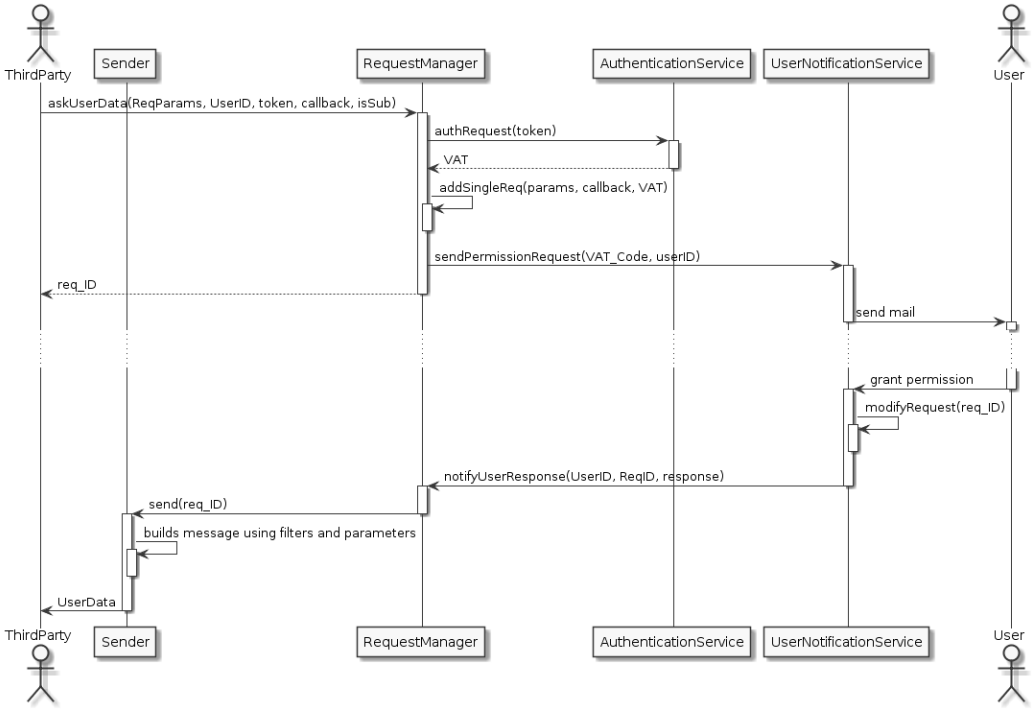


Figure 12: Single User Data Request Sequence Diagram

2.5.4 Group Data Request

In case of group data requests, no user authorization is needed. The only thing that must be checked is if the number of users in the target group is high enough to protect users' privacy. This task is carried out by a dedicated service which is invoked synchronously. The corresponding response shall be then sent to the third party who made the request.

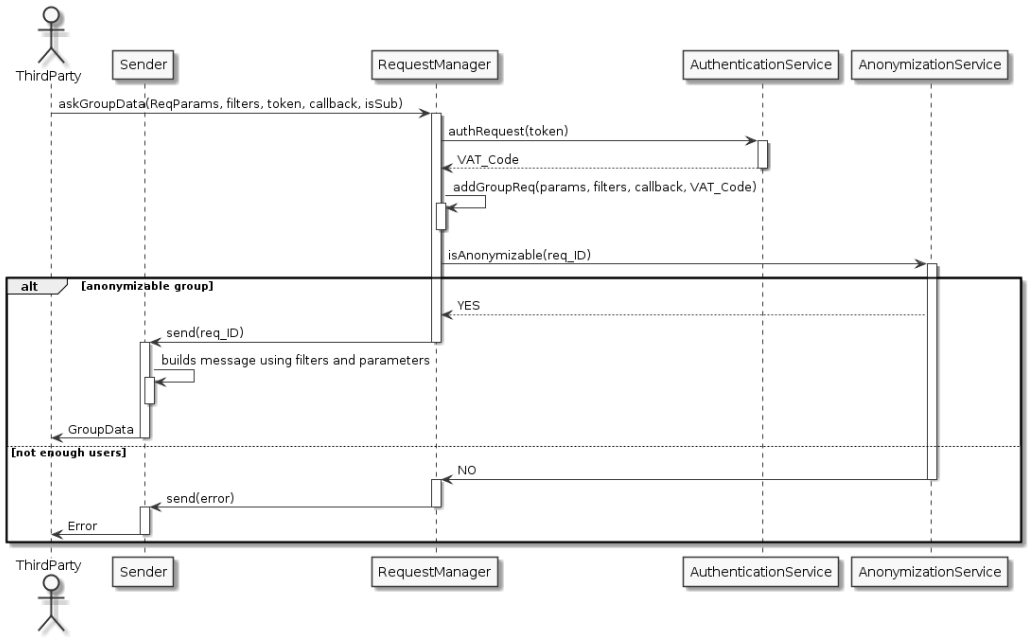


Figure 13: Group Data Request Sequence Diagram

2.6 Component Interfaces

In the following diagram the main methods of the system's components are described in order to explain their interactions. These interfaces and methods are needed whatever the implementation of the underlying component will be, to guarantee that the whole system will meet its functionalities. Asynchronous communication between components is represented with dashed lines and in the receiving components, the *handleMessage()* function is called when a message is received.

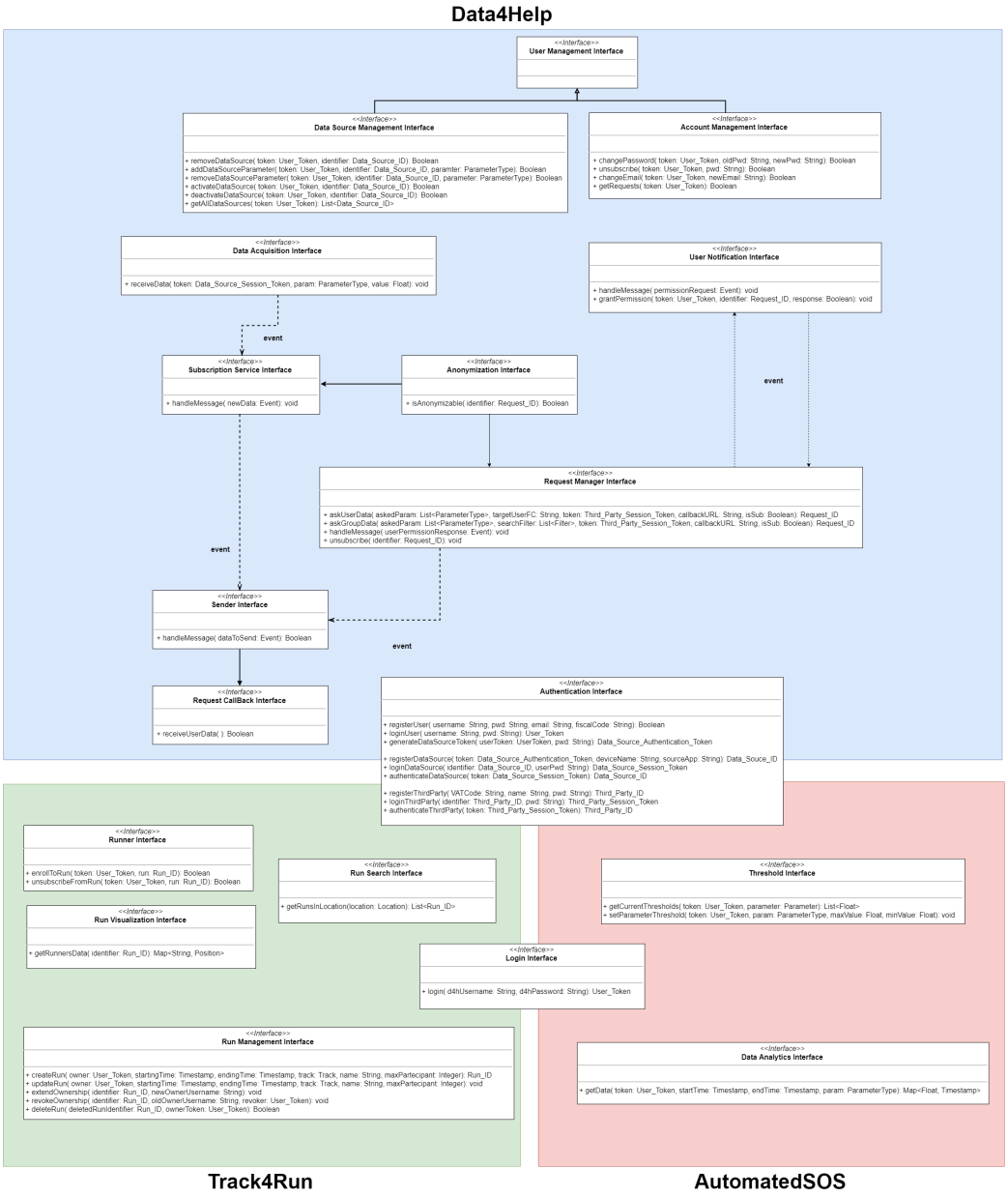


Figure 14: Component Interface Diagram

2.7 Selected Architectural Styles and Patterns

- **Loose Coupling**: the first principle we followed in the design of the whole product is loose coupling, which means that we tried to minimize the dependencies between components of a subsystem and between the three subsystems as well. This means that each subsystem can be developed, deployed and maintained separately, which gives to the system a good flexibility for future development.

- **Stateless Components**: since one of our major concerns is the reliability of our system and how it recovers from failures, our components are designed to be stateless. This means that each component reads and writes on a DB at every step of its work, so that if one of the services is temporarily down, its state can be easily recovered from the DB.

- **Microservice Architecture Pattern**: For Data4Help we decided to implement a microservice architecture pattern. The idea behind a microservice-based architecture is to apply the *SOA* (*Service Oriented Architecture*) pattern inside a big project by breaking it down in smaller pieces that act independently, communicate asynchronously and can be deployed in any fashion and number. This pattern is used in Data4Help because the achievement of high availability and scalability can be easily reached by decoupling the single functions of the system and deploying them independently in large numbers.

  In particular, for this microservice design we decided to adopt the following principles:

  - **API Gateway Pattern**: A single entry point is provided for external access to the services. This gateway will then redirect the incoming calls to the single services. This gives us the possibility to keep together the request authentication management, load balancing and eventually performance testing. Notice that also the gateway can be scaled by adding another one with the same service registry in common.
  - **Server Side Discovery**: This method is a way of keeping track of which are the services available on the system and where their instances can be found. It uses a *Service Registry* at a known

location where new services register themselves when deployed, so that the Gateway/Load Balancer can read the register to know where to redirect requests.

  - **Asynchronous Queuing**: This pattern is used together with the *Event Driven Architecture* pattern to achieve asynchronous and non-blocking communication between services. In our case, we implement asynchronous queuing when using message queues for sharing information and notifications between services. In this way, when a service A calls a service B, it only has to send a message and then it can continue with his job, even if the other service is temporarily unavailable.
  - **Access Tokens**: In order to access any service of the system, external requests must include an access token. This token is used to identify and authenticate the actor who is performing the request.

- **REST**: When asynchronous communication is not possible or desirable, for example during external calls to the system's services, a REST architecture is used to build RESTful APIs that can be accessed by software written in any language. This is another way in which we decouple the functions of our system.

- **MVC**: For AutomatedSOS and Track4Run subsystems, we decided to adopt a standard *Model-View-Controller* monolithic architecture. This gives the possibility to the developers of this subsystems to use well-known commercial solutions to implement these applications.

- **Client-Server Pattern**: The classical separation of the MVC pattern between Model and View is strengthen using the client-server pattern, where there is a client asking for a resource and a server that must provide it. For AutomatedSOS and Track4Run the client is a *thick-client* which takes care of the whole presentation part, whereas in the Data4Help subsystem we have a *thin-client* application for the users, which sits on their browser.

2.8 Other Design Decisions

Some implementation-level solutions have been explored to better understand the problems that could occur during the implementation phase.

In particular, these are some technologies that have been taken into account:

- **NoSQL Databases**: One of the key features of these kind of databases is scalability, and apparently they are very well integrated in many frameworks used for developing microservices. For this reason these kind of databases should be taken into account.
- **JWT**: JSON Web Tokens are a web authentication standard. They are lightweight, cross-platform, encrypted and easy to use. Alternatively, a possible solution for the authentication could be using *OAuth2* servers.
- **Containers**: containers are used a lot in microservices deployment, and should be considered for the implementation of the Data4Help subsystem: they have some security drawbacks with respect to virtual machines or other cloud-based solutions. On the other hand, they are a fast and reliable way to manage many instances in a microservice cluster.
- **Proactive Monitoring**: tools like *Prometheus* enable metric analysis and monitoring inside a microservice cluster. These tools can be easily used to identify a lack of computational power and dynamically scale our application.

3 User Interface Design

User interfaces design has been already shown in the RASD using many mockups.

4 Requirements Traceability

The design choices that were made for the TrackMe system have always kept in mind the fulfillment of the goals and requirements defined in the RASD. Below the reader can find three tables, one for each subsystem, where each requirement is mapped with the component or function that aims to fulfill it.

Table 1: Data4Help requirements traceability matrix

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15
Authentication	✓		✓		✓	✓	✓								
User Management		✓		✓											
Receiver					✓								✓		
User Notification								✓							
Anonymization									✓	✓					✓
Subscription											✓		✓		
Request Management									✓			✓		✓	✓
Sender													✓		
Requests DB											✓				

Table 2: AutomatedSOS requirements traceability matrix

	R16	R17	R19	R20	R21	R22	R23	R24
Login Service	✓				✓			
Third Party SDK		✓			✓			
Monitoring Service		✓				✓	✓	✓
Threshold Setting Service			✓	✓				
Client Application						✓	✓	

Table 3: Track4Run requirements traceability matrix

	R24	R25	R26	R27	R28	R29	R30	R31	R32	R33	R34
Login Service	✓										
Run Creation Management		✓	✓	✓	✓	✓					
Run Search Service							✓			✓	
Enrollment Manager							✓	✓			
Spectator Service										✓	
Third Party SDK									✓		
Client Application											✓

5 Implementation, Integration and Test Plan

5.1 Implementation Plan

For the implementation phase of our project we decided to follow a bottom-up strategy. In particular, starting from the Data4Help subsystem, we will implement each component separately and then perform unit tests on it. By adopting this strategy we incrementally develop working components. As it concerns the implementation order, we decided to start from the Data4Help back-end module, since we think is the most critical one because the other two subsystems work on top of it. In a second time, developer SDKs should be completed before the other two subsystem's back-end development can be carried out. Finally, we can integrate front-ends in all our subsystems. The order could be the following:

1. Data4Help back-end
2. Third Party SDK
3. Data Source SDK
4. Automated SOS and Track4Run back-ends
5. Data4Help front-end
6. AutomatedSOS and Track4Run front-ends

5.1.1 Data4Help Basic Back-End

The first components to be developed should be the ones that guarantee the basic Data4Help back-end functions, which means all the services that are related to data acquisition and data requests. Here is a diagram representing a possible order in which these components should be developed. As already stated, after each development phase a *Unit Test* phase and an *Integration Test* phase should follow, in order to validate the new component and verify its integration with the previously developed system.

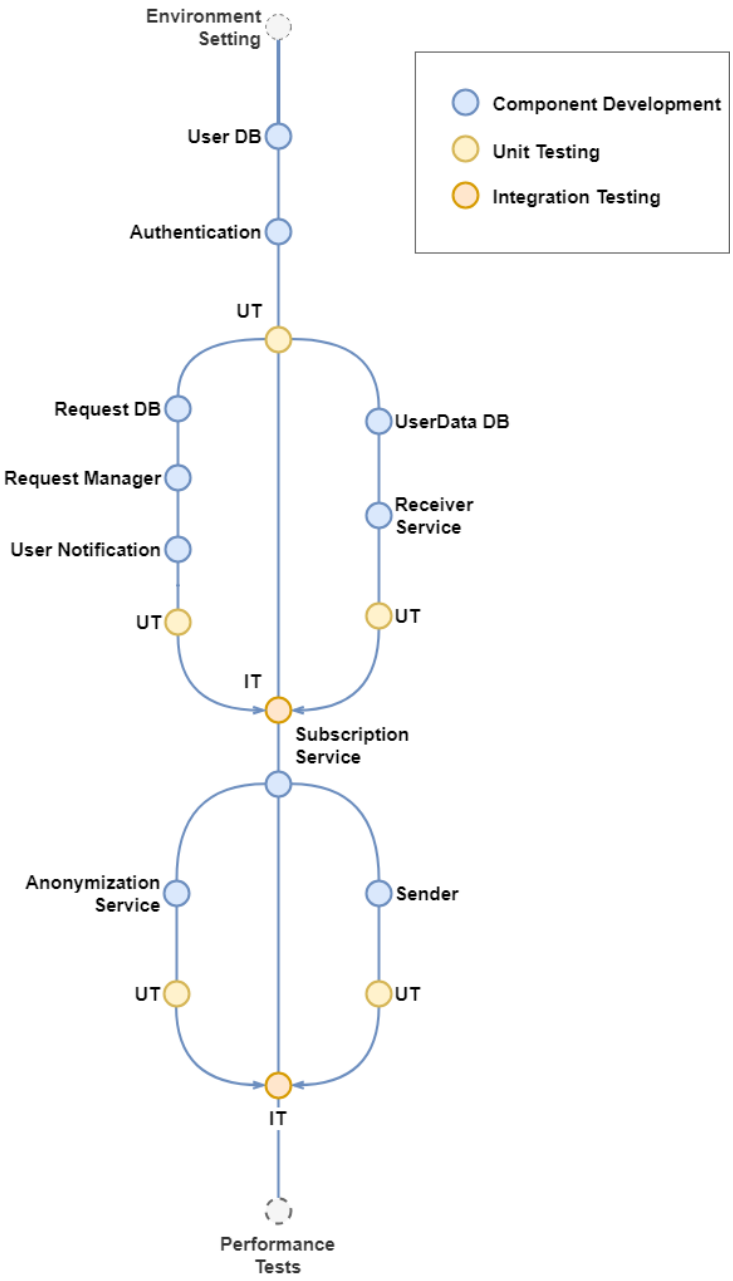


Figure 15: Data4Help Development Plan

5.1.2 Developer's SDKs

The Third Party SDK and Data Sources SDK must be carried out after Data4Help basic back-end is completed and before developing other back-ends. This will guarantee that AutomatedSOS and Track4Run subsystems will be able to properly interact with the Data4Help APIs.

In particular, there is no dependency between the two SDKs, but certainly the Third Party SDK is the most important one in this phase, since Automated-SOS and Data4Help development depend upon it. Anyway the development of these two subsystems can start independently, and we can integrate these SDKs in a second moment.

5.1.3 AutomatedSOS and Track4Run Back-Ends

After the SDKs are developed, the remaining two subsystems' back-ends can be developed in a fully parallel line. Like in the case of Data4Help, an incremental approach is recommended in these development phases, and Unit and Integration tests should be performed at every step.

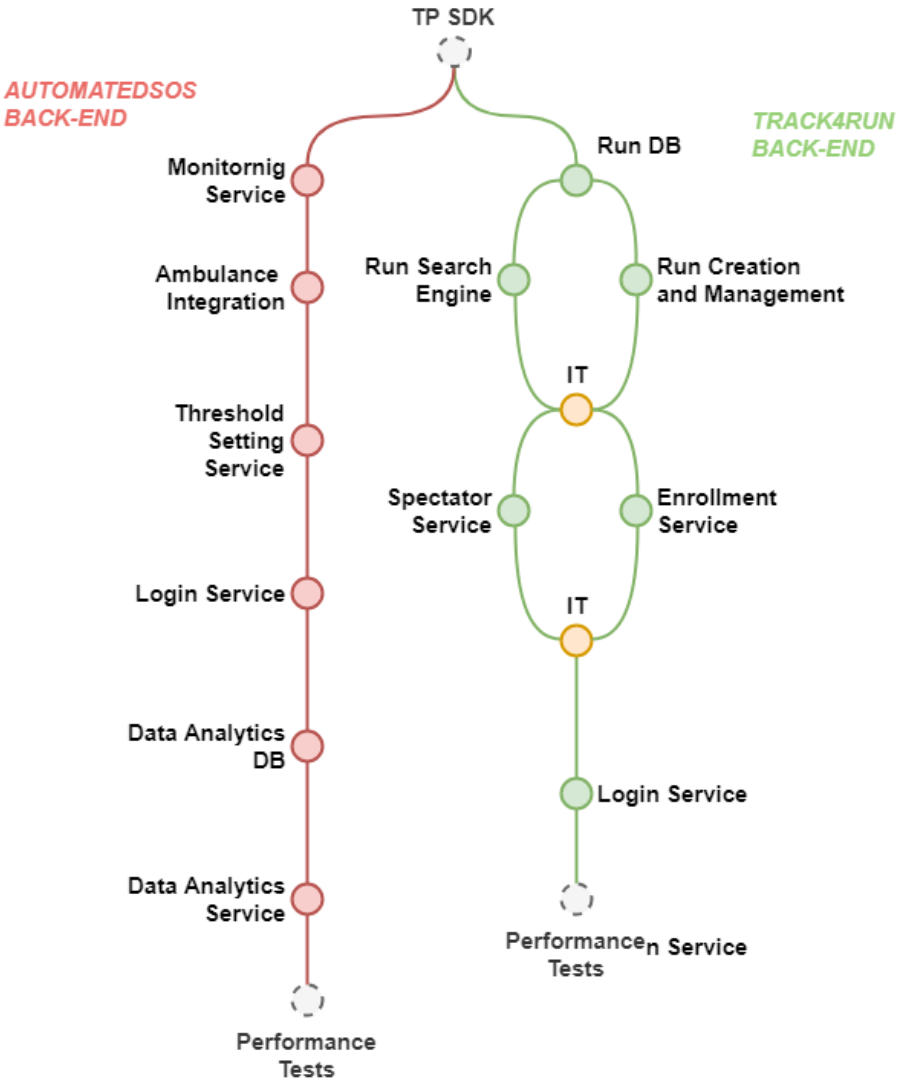


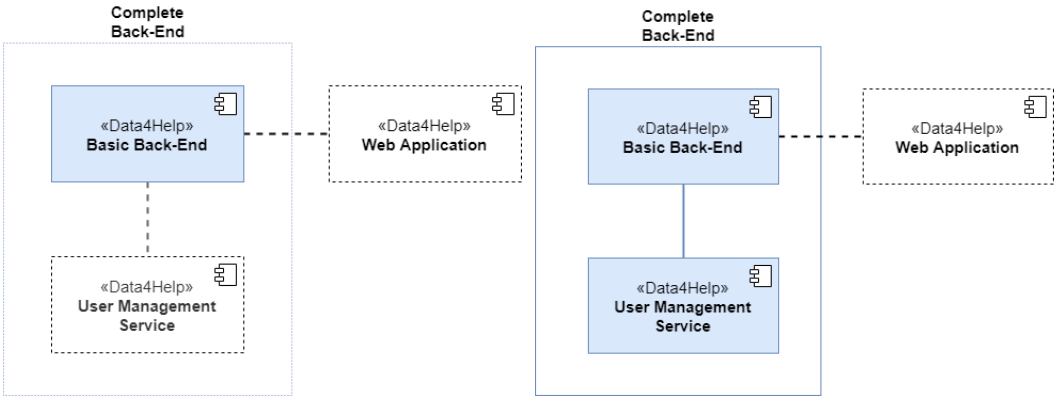
Figure 16: Data4Help Development Plan

5.1.4 Front-Ends and External Services Integration

Finally we can develop the front-ends and integrate the subsystems with the external services. Each front-end is obviously independent from the others, and the external service integration can as well be performed separately for

each subsystem.

- **Data4Help**



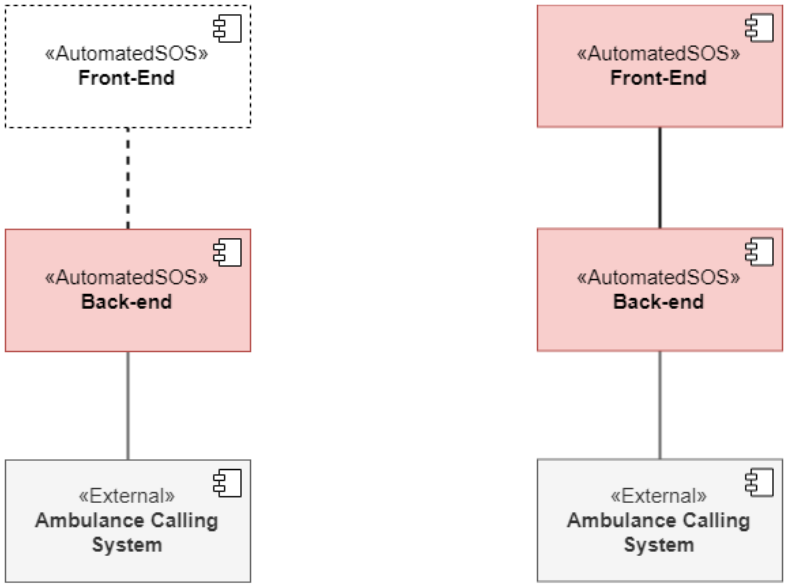
(a) Stage I

(b) Stage II

(c) Stage III

Figure 17: Data4Help Front-End Integration Steps

- **AutomatedSOS**

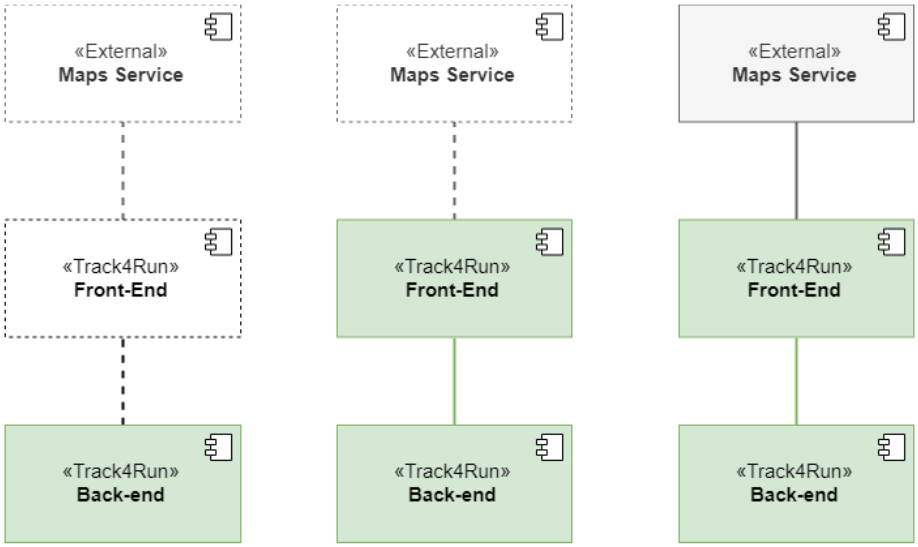


(a) Stage I

(b) Stage II

Figure 18: AutomatedSOS Final Integration Steps

- **Track4Run**



(a) Stage I (b) Stage II (c) Stage III

Figure 19: Track4Run Final Integration Steps

5.2 Integration and Testing

5.2.1 Integration Testing Strategy

With respect to the integration between developed components, we decided to follow a *continuous integration* strategy: as soon as a first version of a component is released, we need to test it and then integrate it with the already existing components of the system. This strategy has been chosen because it enables the developers to have many checkpoints where they can test their product and makes it easy to find bugs early in the development of the system, so that if things break, they break small.

Once each back-end have been completely developed and integrated, it's also highly recommended to carry out a system testing process. In particular:

- **Performance Testing:** for each subsystem, we need to monitor the responsiveness of the services with respect to incoming requests. In particular, Data4Help should be very fast in managing new data samples to guarantee a fluid service to the third-parties. In addition, AutomatedSOS needs to be extremely fast in detecting and responding to an emergency due to the 5 seconds time constraint. Lastly, Track4Run user experience is heavily influenced by the back-end performances, which should be for this reason flawless.
- **Load Testing:** after the performance testing, is suggested to continuously feed each subsystem with an increasing number of requests. This is done to assess the performances and availability of each subsystem in case of unusually high volumes of incoming requests. This phase also tells us if the system can handle malicious attacks which try to exploit memory leaks, buffer overflows and similar bugs.
- **Stress Testing:** finally, in order to test if the subsystems are able to recover from unexpected failures, we should simulate them in a controlled manner.

5.2.2 Entry Criteria

As stated above, continuous integration forces the testing phase to start as soon as a component is merged with the main product. However, it is necessary to be sure that a correct environment is setup before integration can be effectively carried out. In particular, in case of Data4Help we need the following components:

- *Microservices Containers*
- *API Gateway*
- *Message Queue Server*
- *Service Registry*
- *DataBase Managment System*

- *Performance Monitor*

For AutomatedSOS and Track4Run instead we need the web server and firewalls to be correctly set-up and the *Third-Party SDK* to be available and already tested against the Data4Help request APIs.

5.2.3 Elements to be Integrated

As seen in the development plan, we can divide our components in groups considering the existing dependencies between them. The following diagrams are meant to clarify these dependencies:

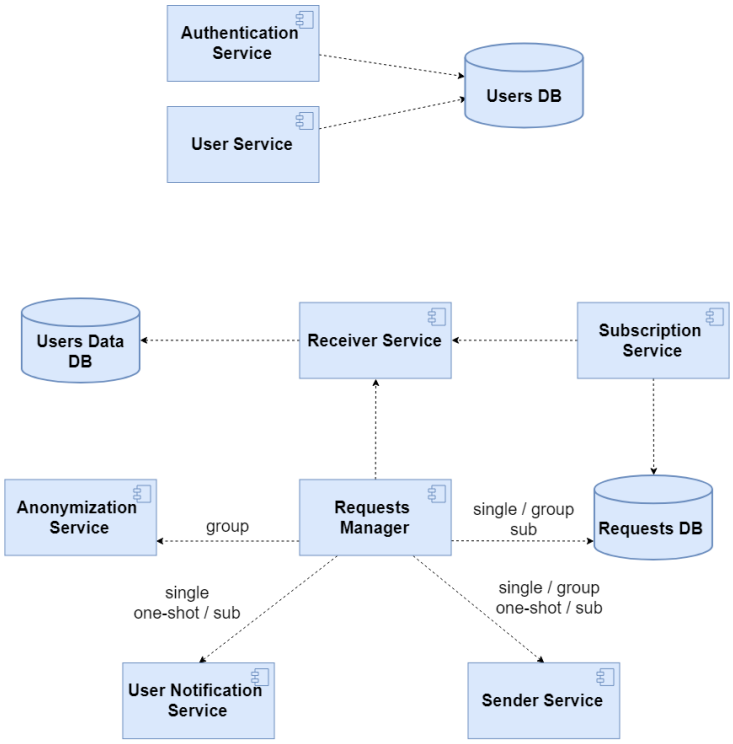


Figure 20: Data4Help Components Dependencies

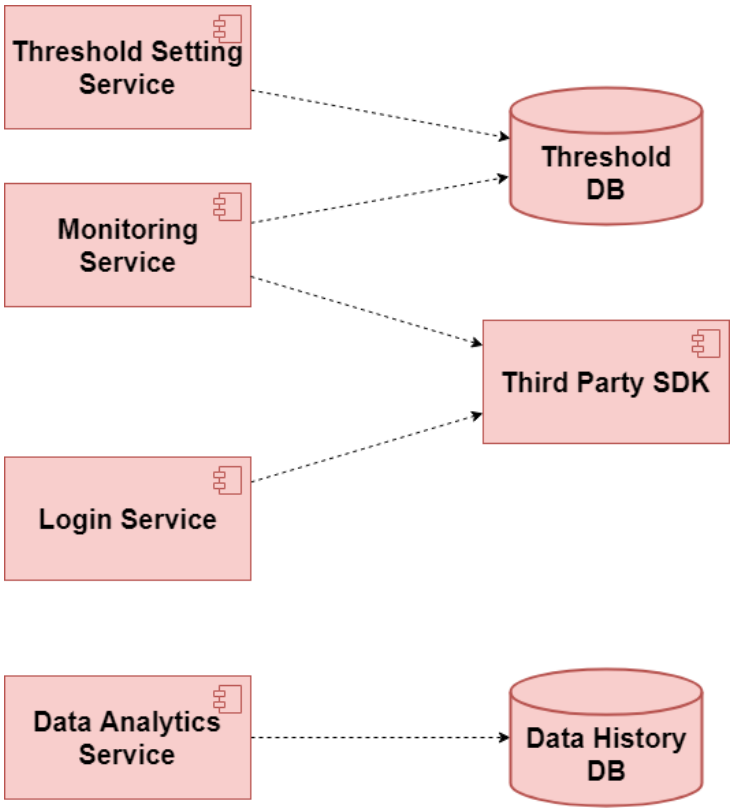


Figure 21: AutomatedSOS Components Dependencies

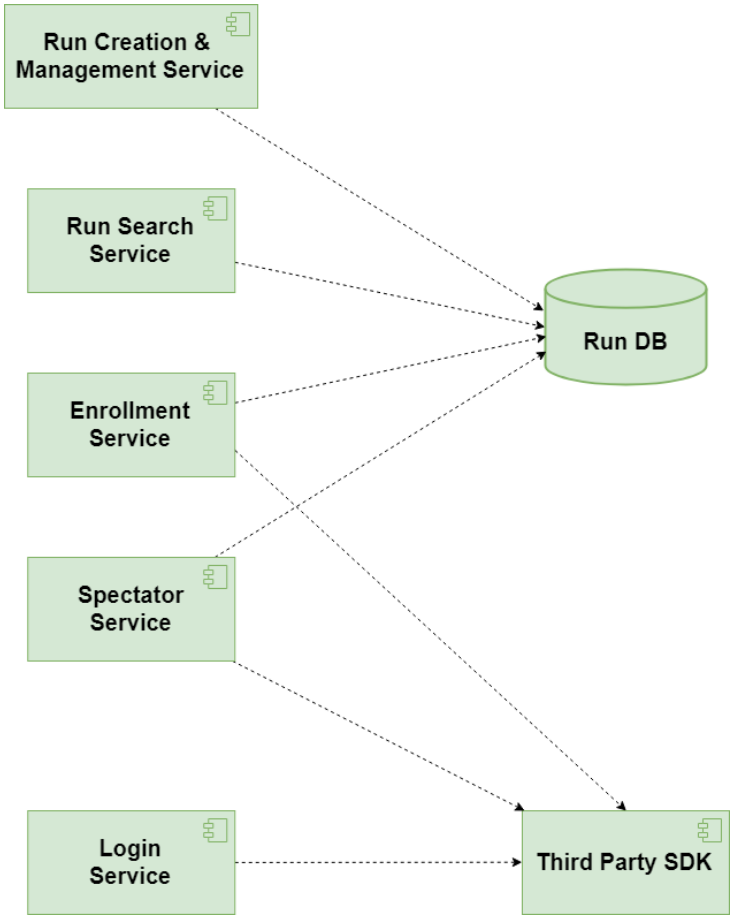


Figure 22: Track4Run Components Dependencies

5.2.4 Sequence of Component/Function Integration

Since we adopted a continuous integration strategy, the integration phase runs in parallel with the implementation phase, which is carefully described in the previous diagrams.

6 Effort Spent

Marco Gelli	
Component Brainstorming	4 hours
Requirements Traceability	4 hours
ER Diagram	2 hours
Interfaces	4 hours
Component Description	5 hours
I&T Plan	4 hours
ER Diagram	3,5 hours
Document Review	12 hours

Alvise de'Faveri Tron	
Component Brainstorming	4 hours
Requirements Traceability	5 hours
ER Diagram	2 hours
Interfaces	4 hours
Microservices Deployment	4 hours
Component Description	5 hours
I&T Plan	4 hours
Patterns Explanation	3,5 hours
Document Review	7 hours

Andrea Biscontini	
Component Brainstorming	4 hours
Requirements Traceability	5 hours
Interfaces	4 hours
Traceability Matrix	5 hours
Interfaces	3,5 hours
Document Review	11 hours

7 References

- *Requirement Analysis and Specification Document*
- Assessment: *Mandatory Project Assignment AY 2018-2019.pdf*
- Microservices Patterns
- OAuth2 okta server
- Containers, Docker documentation
- Cloud Computing Patterns
- REST, Spring Boot documentation
- NoSQL, MongoDB documentation